AS

(C) XPESP / ELSEVIER
AN - XP004339628
PAN - S0167-739X(01)00074-7
IRN - ISSN 0167-739X
PUB - Future Generation Computer Systems
VOL - 18
NR - 4
PG - 491 - 499
PD - 2002-03-00
AU - Geoffray P
TI - OPIOM: Off-Processor I/O with Myrinet
AB - As processors become more powerful and clusters larger, users will exploit this increased power to progressively run larger and larger problems. Today's datasets in biology, physics or multimedia applications are huge and require high-performance storage sub-systems. As a result, the hot spot of cluster computing is gradually moving from high performance computing to high performance storage I/O. The solutions proposed by the parallel file-system community try to improve performance by working at the kernel level to enhance the regular I/O design or by using a dedicated Storage Area Network like Fiber Channel. We propose a new design to merge the communication network and the storage network at the best price. We have implemented it in OPIOM with the Myrinet interconnect: OPIOM moves data asynchronously from SCSI disks to the embedded memory of a Myrinet interface in order to send it to a remote node. This design presents attractive features: high performance and extremely low host overhead.
TXT - PII S0167739X01000747 S0167-739X(01)00074-7]> This work has been realized with the help of Sycomore-Aerospatiale Matra in France, and Jack Dongarra and Rich Wolski at the University of Tennessee, USA. OPIOM: Off-Processor I/O with Myrinet Patrick Geoffray 1 1 URL: http://www.myri.com . patrick@myri.com Myricom, Inc., 685 Emory Valley Road, Suite B, Oak Ridge, TN 37830, USA As processors become more powerful and clusters larger, users will exploit this increased power to progressively run larger and larger problems. Today's datasets in biology, physics or multimedia applications are huge and require high-performance storage sub-systems. As a result, the hot spot of cluster computing is gradually moving from high performance computing to high performance storage I/O. The solutions proposed by the parallel file-system community try to improve performance by working at the kernel level to enhance the regular I/O design or by using a dedicated Storage Area Network like Fiber Channel. We propose a new design to merge the communication network and the storage network at the best price. We have implemented it in OPIOM with the Myrinet interconnect: OPIOM moves data asynchronously from SCSI disks to the embedded memory of a Myrinet interface in order to send it to a remote node. This design presents attractive features: high performance and

extremely low host overhead. Myrinet SCSI Storage Disk I/O Linux Cluster 1 Introduction The availability of powerful microprocessors and high-speed networks as commodity components is making clusters an appealing solution for cost-effective high performance computing. However, the bottleneck for users' applications tends to shift from the computation and the communication sides to the I/O domain: the problem sizes are bigger and bigger and the time to load datasets into the cluster work pool and write the results to disks cannot be neglected any more. The new generation of commodity components like storage controllers and high-speed networks can be efficiently used to break some architectural limitations inherited from the past, while keeping the price/performance ratio as low as possible. Our research effort improves a basic feature for the usage of parallel I/O on clusters by removing bottlenecks. In ---Section--- 2 , we present the motivation of this work, one current limitation of the parallel I/O design for clusters, and some related work that tries to improve it. We propose a new design in ---Section--- 3 , describing our contribution and detailing the issues that occurred during the implementation. Then, ---Section--- 4 shows the results of some experimental benchmarks to highlight the benefit of our work for parallel I/O implementations. Finally, the conclusion in ---Section--- 5 summarizes our work and presents the short and medium term perspectives of our project. 2 Motivation Today's clusters are larger and more powerful than ever before. They start to be used to face some Grand Challenges in genomics or nuclear simulations, or even for intensive multimedia applications like Video-on-Demand (VOD). The datasets used in these contexts are very large, and require the I/O sub-system to be as efficient as the computation or the communication components. There are two ways to achieve high-performance I/O in a cluster environment: To use a dedicated Storage Area Network (SAN) like Fiber Channel [1] to connect the storage units, the SCSI disks, to all of the nodes of the cluster. This choice is quite expensive as clusters already include a high-speed interconnect to support message passing. One effort [2] focused on providing an MPI implementation on top of Fiber Channel, without a real success. To use disks attached locally to each node of the cluster with a parallel file-system: the interconnect network is used to send I/O requests to remote nodes and to receive the data from them. It seems to be the solution chosen by the research community on cluster computing in order to preserve the cost-effective advantage of clusters. There are a lot of prototypes and production-quality parallel or distributed file-systems available worldwide like the well-known NFS [3] , GPFS for the IBM SP [4] or more recently PVFS [5] . All of these contributions present improvements and new features on parallel disk access, cache management, load balancing, fault-tolerancy, etc. Technology is improving rapidly and the availability of efficient storage buses such as SCSI Ultra160 or ATA-100 coupled with high-speed networks like Myrinet raises the expectation of parallel file-system implementations using these components. However, such parallel file-systems use a significant amount of resources, like processor cycles or memory on the nodes that host disks and it is not unusual to use dedicated nodes to act as I/O servers for the rest of the cluster. To process I/O requests from a remote process and serve the data requested is still a very heavy charge on today's machines. 2.1 Remote read The Remote Read is a basic operation of parallel file-systems. As the data is balanced along disks on different nodes, it is necessary to send an I/O request via the interconnect network to the remote nodes to ask them to read some data locally before sending it back to the client. Fig. 1 Fig. 1 Data movements on the server side in the Remote Read operation between a SCSI controller and a Myrinet interface. illustrates the data movements in a node processing an I/O Read request on a SCSI sub-system from a remote client via Myrinet. The data path is particularly long and uses several hardware components that can be possible bottlenecks. We can distinguish three steps: A) In response to the SCSI commands posted from the driver in the kernel, the disks read the data and transfer it over the SCSI bus to the SCSI controller, and the latter copies it to the kernel memory space via the PCI bus and the memory bus using a DMA engine. B) After catching the interrupt from the SCSI controller to indicate the completion of the SCSI commands and the DMA copy, the kernel updates the buffer cache entries and copies the data into the user-level application memory space by crossing the memory bus two times and using the processor to perform the copy. C) Finally, the data is sent to the remote node directly from the application memory space with the zero-copy communication protocol available on Myrinet. During this operation, the data crosses the memory bus and the PCI bus to the Myrinet embedded memory before being pushed on the link. We can see in [6] that the PCI bus and the memory bus are more of a limit on the throughput of this Remote Read operation than the SCSI bus or the Myrinet link. In fact, each unit of data passes one time on the SCSI bus or the Myrinet link, two times over the PCI bus and four times over the memory bus. As the peak capacity of each of these components tends to converge, the storage system and the interconnect are not efficiently exploited. 2.2 Related work Other projects have a similar approach to improve the Remote Read operation by reducing the critical path. 2.2.1 Linux raw I/O The raw I/O concept tries to remove step B in Fig. 1 . It enables access to the storage device directly from the user-space application by bypassing the kernel space, thereby avoiding the buffer cache overhead and saving a memory copy. Raw I/O is available as a patch to the Linux kernel and, after strong pressure from the database community who wanted to minimize the kernel cost and allow the low-level storage device management at the application level, will be included in Linux kernels 2.4.x. 2.2.2 Intelligent I/O (I2O) I2O [7] is an architecture designed to eliminate the bottlenecks by using dedicated I/O processors that offload the main processor to handle the movements of data, the interrupts, the flow control, etc.

In the I2O model, the I/O operations are messages exchanged between I2O devices, e.g. between an I2O SCSI controller and a dedicated i960 or directly between an I2O SCSI controller and an I2O ATM network interface. The I2O are completely autonomous and are able to support asynchronously a large part of the I/O processing. I2O has difficulties to entice the market, certainly due to the very high cost of the I2O devices and the development tools, as well as the close-minded nature of the specifications. 2.2.3 Network Attached Storage (NAS) NAS is an important ongoing work. NAS differs from the SAN-like Fiber Channel by providing a file system functionality instead of a fixed-size ---block---oriented interface. NAS is based on SAN for reliable and high-performance communication support, but the rich interface of the NAS removes the limitation of SAN in handling the storage devices as a single hardware unit shared by all of the clients. Gibson and Meter [8] presents a large state-of-art. 3 Contribution The data path between the storage controller and the network interface passes through the host memory despite the fact that the data is not processed by the main processor before being sent to the I/O request emitter. The data goes through the host because of system constraints: the interactions with a local storage controller are traditionally operated from a user application and the communication interface of the NIC assumes the data to be present in the main memory at the beginning of a send. It would be very efficient to drastically shorten the data path by moving data directly from the storage controller to the network interface over the PCI bus (Fig. 2 Fig. 2 Data movements on the server side in the Remote Read operation with OPIOM.) like the I2O peer-to-peer operations. It is technically possible as all of these devices provide DMA engines and embedded memory on the PCI address space. With DMA engines, a PCI device is able to write or read data in main memory but also on any other PCI memory addresses. However, only a few devices provide enough memory on-board and the flexibility to access this memory and use DMAs. We have chosen to work with Myrinet interfaces for the software openness, the possibility to modify the firmware, the amount of embedded memory (up to 8 MB for the latest generation) and the performance. The Gigabit Ethernet Alteon also presents some good characteristics but the performance and the flexibility of the firmware are limited. We have designed an interface to handle the Remote Read operation with the shortest possible data path between SCSI devices and a Myrinet interface: OPIOM. 3.1 OPIOM OPIOM stands for Off-Processor I/O with Myrinet. OPIOM is an implementation for Linux of the optimization previously described. OPIOM supports all of the SCSI controllers supported by Linux and Myrinet as the network interface. However, OPIOM is generic enough to easily support any other network interfaces providing the basic requirements: a PCI memory space and the possibility to send/receive packets from/to buffers on this embedded memory. The choice of Linux is obvious as we need to know how the kernel processes I/O requests and to find the best point to insert our code. Linux is open-source and allows us to understand the I/O code and, eventually, to slightly modify it to serve our purpose. The implementation described below is organized into two main parts: one ---section--- is related to the interactions with the storage controller, in our case the SCSI controller; and a second one dedicated to the usage of the Myrinet network interface. 3.2 Implementation 3.2.1 SCSI side The storage part represents the active core of OPIOM. OPIOM is composed of a user library and a kernel module that will insert a new SCSI service at the top of the Linux SCSI stack (Fig. 3 Fig. 3 SCSI stack in the Linux kernel 2.2.x.). 3.2.1.1 Linux SCSI stack The design of the SCSI stack in the Linux 2.2.x. kernels is particularly elegant: its decomposition into three layers separates the SCSI services (SD for disks, SR for CD-ROMS, ST for tapes and SG for generic devices) from the code of low-level drivers by an abstraction layer, the SCSI middle layer. The SG module presents interesting capabilities: it is used to develop and test new SCSI services before their integration in the kernel. SG can post some SCSI requests to the middle level layer to be processed. The OPIOM module is very similar to SG in the way that the user-level application posts asynchronous operations via ioctl( ) calls. The OPIOM module generates the corresponding SCSI commands and integrates them into SCSI requests that are passed to the Linux SCSI middle layer. The SCSI requests are composed of three fields: The SCSI command ---block--- that contains the command sequence will be interpreted by the SCSI device. It includes the SCSI operation code, the number of physical ---blocks--- to process, etc. The virtual memory address of the target buffer in the kernel space. The controller will copy the data to this buffer by DMA. A completion function that will be called by the interrupt handler when the SCSI command will be completed and the kernel notified by a hardware interrupt from the SCSI controller. The SCSI middle layer merges the SCSI requests issued by all of the SCSI services in the upper layer. It is then possible to use OPIOM to access a disk at the same time as the system for regular I/O. The middle layer can also rearrange the SCSI requests using an elevator algorithm in order to minimize the movement of the heads of the disks. At the completion of a request, the middle layer checks the error code, eventually tries to recover errors and calls the completion function associated with the request. This call-back mechanism allows OPIOM to manage in a fully asynchronous way the processing of SCSI requests. 3.2.1.2 Local file-systems The applications handle file descriptors, not numbers of physical ---blocks--- on disks. Therefore, it is necessary for the OPIOM module to translate the application language, the file descriptors, into SCSI command language, physical ---blocks--- on disks. The file-systems supported by Linux are integrated in a file-system abstraction called the Virtual File System (VFS). The VFS provides functions to map a logical ---block--- to a physical ---block--- and to translate a position in a file to the reference of the physical ---block--- on the disk. OPIOM supports any file-system compliant with the Linux VFS, even with software RAID functionality. We must also address the issue of fragmentation. This issue is related to the local file-system usage: a SCSI command processes contiguous ---blocks--- on disks; however, the data is fragmented on the medium as a result of creation and destruction of files. OPIOM needs to decompose a global operation on a file, called an OPIOM slot , into one or more OPIOM requests that will be associated to SCSI requests. The completion of an OPIOM slot would mean the completion of all of the OPIOM requests composing it. 3.2.1.3 Memory address conversion A major difficulty exists as a result of the design of the SCSI stack in Linux: a SCSI request includes the virtual address of the target buffer allocated in the kernel space. This virtual address is converted in the low-level driver corresponding to the SCSI controller into a physical address usable by the DMA engine component. The problem is that the target buffer with OPIOM is on the Myrinet board, not in the kernel memory space. If OPIOM posts a SCSI request with the PCI address of the buffer in the Myrinet SRAM, the function virt_to_phys( ) in the low-level driver will return a false address somewhere in the kernel space and a DMA operation to such an address will corrupt and crash the machine. We are obliged to slightly modify the behavior of the virt_to_phys( ) function in order to avoid meaningless translation for physical addresses. The Linux kernel maps the physical host memory into the kernel space starting at the constant ---PAGE---_OFFSET . The BIOS maps the memory areas of PCI devices in the kernel address space, usually at the end of it. The addresses of buffers on the Myrinet board will be in this PCI area. In this context, to translate a kernel virtual address into a physical memory address, one only needs to subtract the value ---PAGE---_OFFSET from the virtual address. As the virtual addresses translated by this method are always in the kernel space, we can extend the virt_to_phys( ) function: if (address < ---PAGE---_OFFSET) return (address + ---PAGE---_OFFSET); else return (x ---PAGE---_OFFSET); We can then avoid a translation by removing the value ---PAGE---_OFFSET from the physical address. By this hack, we can avoid the modification of all of the low-level SCSI drivers supported by Linux. Thus, it is possible to transparently manage in a fully asynchronous way I/O operations from SCSI disks to buffers on any PCI device. The OPIOM kernel module that handles the interactions with the Linux SCSI stack can be dynamically loaded. The intrusivity is limited to the three lines added to the virt_to_phys( ) function. No modifications will be needed with the Linux kernel 2.4.x because of a change in virt_to_phys( ) . 3.2.2 Myrinet side Once the data is in a buffer in the Myrinet memory, we need to be able to send it to another node. The Myrinet interface provides the two functionalities required: It embeds a large amount of memory, directly accessible from the PCI bus. It can send a packet on the link from a buffer on its own embedded memory. The second functionality, called a send buffer-in-place , has been implemented by modifying the GM [9] firmware for Myrinet. The send

buffer-in-place is similar to a regular send without the first step
that consists to copy the message from the host memory to the Myrinet
SRAM by DMA. GM's packets are limited to 4 KB and the header and the
body of each packet, contiguous in the Myrinet memory, are written to
the link in one DMA operation. This model does not allow to send a
packet with the header in one place and the body in another one. As
the number of memory areas to send packets is limited at 2, it is not
possible to stop the sending activity on the Myrinet card during an
OPIOM operation to copy the data directly to the right place. A simple
solution consists in copying the data from an OPIOM buffer in the
Myrinet SRAM to a send buffer in the logical structure of the GM
firmware. The next generation of GM will manipulate headers and bodies
separately, so this problem will disappear. The cost of this extra
copy depends on the speed of the Lanai, the processor embedded on the
Myrinet board. The Lanai 9 at 133 MHz can copy word-aligned data at
266 MB/s. GM implements a blocking system call to sleep, while waiting
for an interruption from the board to notify an event. While the
process is sleeping, it is removed from the list of runnable processes
in the scheduler and effectively does not use the CPU. This is a very
important feature in the context of parallel file-systems where
computation nodes on a cluster are also used for I/O. 4
Experimentation We have conducted experiments with OPIOM to validate
the implementation and measure the performance gain versus the regular
I/O implementation. 4.1 Platform The platform is composed of two Linux
boxes, one server and one client: Server: machine Dell PowerEdge 2300,
Pentium II 450 MHz, 256 MB, PCI 32 bits/33 MHz, running Linux 2.2.17.
The kernel on this machine includes the virt_to_phys( ) function
modification. This machine hosts also a storage sub-system composed of
an Adaptec AIC-7890 Ultra2 SCSI host adapter (one bus at 80 MB/s of
theoretical peak throughput) used by the aic7xxx low-level driver and
6 SCSI disks Seagate ST34573LC Ultra2-LVD at 7200 rpm, all of them on
the unique SCSI bus. The disks are connected by a hot-plug SCSI
backplane Dell. The Myrinet interface is a PCI64B (Lanai 9 at 133 MHz)
with 8 MB of embedded memory. The communication interface for Myrinet
is GM 1.3.1 with the send buffer-in-place addition. The file-system
used on the disk is the default Linux ext2fs, compiled with a logical
---block--- size of 4K. The Myrinet card and the SCSI controller are
on the same PCI ---segment--- without other PCI devices. Client : PIII
500 MHz, 512 MB, PCI 32 bits/33 MHz, running Linux 2.2.17. The Myrinet
interface is a PCI64B (Lanai 9 at 133 MHz) with 2 MB of embedded
memory. The communication interface for Myrinet is GM 1.3.1 without
any modification available on Myricom's website. The Myrinet
interfaces are connected with an 8-port switch. Despite the fact that
these boards can use a link at 2+2 GB/s, the switch is not compliant
with this new link speed and forces the interfaces to use the former
rate, 1.28+1.28 GB/s. 4.2 Tests We ran three series of tests on our
platform. The first experimentation aims to validate and evaluate the
performance of the OPIOM core interacting with the storage sub-system,

and the second uses all of the components involved in the Remote Read operation to implement the movement of data from disks to a remote node. The third test shows some preliminary results in the context of a parallel video server application. For all of these tests, the dataset is stripped along the 6 disks (RAID 0) at the application level, it provides more flexibility to investigate several stripping unit sizes without regenerating the dataset. Actually, each disk contains a 1 GB file filled with marks and stamps used to check the validity of the data on the client side for the copy test. 4.2.1 Local read This test tries to read data as quickly as possible from disks at destinations of buffers in the process memory space with the regular I/O implementation or in the Myrinet SRAM with OPIOM. With regular I/O, the test uses one thread per disk to simulate asynchronous reads with the synchronous functions of the C library. Each thread reads 128 KB of data from its corresponding disk. The activity of I/O threads is managed by another thread that drives the test and insures the order of data gathering. This allows us to operate several I/O requests concurrently and to be able to compare the results with OPIOM. With the latter, we use two buffers of size 128 KB on the Myrinet board up to 4 disks, and four buffers after that. There is only one process that posts OPIOM read requests and waits for their completion. This benchmark is useless from a practical point of view, as the data on the Myrinet memory is not checked or used. However, it is a very stressing benchmark for the storage sub-system. Fig. 4 Fig. 4 Performance of local read with OPIOM and regular I/O. illustrates the performance difference of the local read operation with OPIOM and the regular I/O implementation. The SCSI Ultra2 bus that can theoretically reach 80 MB/s seems to saturate at 72 MB/s, even with more than 6 disks on the bus. The throughput per disk of 14 MB/s is coherent with their specification. New generation 10K rpm disks have been measured at 25 MB/s. We can see that OPIOM and regular I/O performance are similar and linear up to 4 disks. At this point, the regular I/O implementation tends to a maximum throughput of 64 MB/s. However, this is not the maximum available bandwidth, as OPIOM progresses up to 72 MB/s. Fig. 5 Fig. 5 CPU usage of local read [illegible]th OPIOM and regular I/O. emphasizes the significance of OPIOM. I[illegible] shows the CPU usage monitored using top during these local read ope[illegible]ation tests. The processor usage with OPIOM is very low (3% [illegible]aximum for 6 disks) in contrast to the CPU utilization with the regular I/O code that increases linearly with the number of disks and is almost using all of the machine with 6 disks at 88%. This CPU usage is due to the memory copy between the kernel space and the user-level application. This test also permits us to validate the interaction with the Linux SCSI stack as the benchmark has run during 2 consecutive days and read more than 10 TB of data without any corruption. 4.2.2 File copy Pleased by these promising results, we have implemented a high-performance copy benchmark where the data is read from disks and streamed to a remote node via Myrinet. The test code is based on a pull architecture for flow control reasons. Thus, the next piece of data is not sent to the client before the reception of a small message from this client indicating that the buffer is ready to proceed. We used the same code to read the data as in the previous test and the client side uses two 128 KB size buffers to pipeline the reception of the packets from the server. Thus, the granularity along the pipeline is 128 KB. Fig. 6 Fig. 6 Performance of copy with OPIOM and regular I/O. shows very distinctly an improvement using OPIOM compared to the performance using regular I/O. We can also see that the aggregate throughput is cut at 45 MB/s: this limitation is certainly due to the PCI bus at 60 MB/s (120 MB/s half-duplex) minus some bus arbitration overhead. By its unique travel on the PCI bus, OPIOM is not sensitive to this limitation and can exploit the maximum of the storage system. The CPU load results in Fig. 7 Fig. 7 CPU usage of copy with OPIOM and regular I/O. confirms the very low host overhead of OPIOM, less than 3% of a Pentium II at 450 MHz to read data from disks and send it to a remote node at the rate of 72 MB/s. In this case, the only limitation comes from the SCSI bus. The CPU load measured during the test with the regular I/O implementation is not as high as in Fig. 5 because the throughput is reduced by the PCI bus and so the processor does not have to copy as much data. 4.2.3 Video server prototype This test aims to show the interest of OPIOM in real applications where the read patterns are non-linear. In this case, the displacement of the heads of the disks is time consuming and the read-ahead algorithm of storage units much less efficient. We use in this experiment a prototype of a parallel video server as described in [6] : a node serves a video client by requesting data from all of the other nodes, re-assembling the stream before sending it via the distribution network. The different video streams are non-contiguous in the local file-systems. In our experiments, only the server node has the requested data. We measure the latency between the read request from the client node and the delivery of requested data by the server node (transactions of 128 KB). We can see in Fig. 8 Fig. 8 Latency and jitter for 128 KB ---block--- with the video server prototype using OPIOM and regular I/O. that the latency of the read requ sts increases with the number of streams, as expected. However,   PIOM seems to support fairly well the load increase and the variation   this latency stays very small. On the contrary, the regular I/O ir plementation presents unstable performance, with a very large range of variation and a quick degradation of the latency of the read requests under load increase. This bad behavior requires additional buffers and limits the number of streams concurrently supported, decreasing the price/performance ratio of the video server. 5 Conclusions and perspectives Parallel I/O is a very important research domain for high performance cluster computing. Today's clusters cannot be used at the maximum of their capacity because of disappointing I/O performance compared to computational power. We have designed a basic interface to optimize the data movement between disks and an intelligent network interface with Linux.

Our implementation with SCSI and Myrinet, OPIOM, provides high-performance throughput and very low host overhead as well as a UNIX-like transparent I/O library. This tool can be used as a basic Remote Read functionality for parallel file-systems or MPI-I/O implementations where the I/O nodes and the computation nodes can be the same and where the communication interconnect can be used as an SAN, reducing the total cost of clusters. We plan to extend our experimentation with higher-level machines, 64 bits/66 MHz PCI bus and Ultra3 SCSI bus, in order to saturate the resources that limit the regular I/O model, the memory bus and the processor. The next development around OPIOM will provide access to IDE disks, as the way Linux handles them is very similar to the SCSI devices. An emulation mode is already present in the Linux kernel to handle IDE controllers via the SCSI stack. The Write operation with OPIOM is also fairly easy to implement as the data path is the same as for the Read operation, except that the DMA engine of the SCSI controller would read from the Myrinet board to the SCSI device. A buffer cache invalidation mechanism would be needed in this case, as the consistency cannot be guaranteed and the data path avoids the host. We also plan to integrate OPIOM into the next GM releases, with some dynamic Myrinet buffer management and an optimized OPIOM send/receive operation. [1] F.C.I.A. (FCIA), Fiber Channel, WWW, http://www.fiberchannel.com . [2] Finisar, M.SoftTech, NCSA, High-performance solution uses FC, VIA, and MPI, Headlines, NCSA, November 1998. http://access.ncsa.uiuc.edu/Head lines/981106.Finistar.html . [3] H. Stern, Managing NFS and NIS, O'Reilly & Associates, Inc., 1991. [4] Corbett P.F. P NFS and NIS, O'Reilly & Associates, Inc., 1991. [4] Corbett P.F. Parallel file systems for the IBM SP computers IBM Syst. J. 34 2 1995 222 248 [5] P.H. Carns, W.B.L. III, R.B. Ross, R.Thakur, PVFS: a parallel file system for Linux clusters, in: Proceedings of the 2000 Extreme Linux Workshop, 2000. [6] A. Bonhomme, P. Geoffray, High performance video server using myrinet, in: Proceedings of the First Myrinet User Group Conference (MUG 2000), Lyon, France, 2000. [7] I.S.I. Group, Intelligent IO (I2O, WWW, http://www.i2osig.org). [8] Gibson G.A. Meter R.V. Network attached storage architecture. Commun. ACM 43 11 2000 37 45 [9] M.GlennBrown, The GM API, http://www.myri.com/GM/doc/g m_toc.html , December 1998. Patrick Geoffray received his PhD from the University of Lyon (France) in 2001. He is currently a senior programmer at Myricom in the branch office located in Oak Ridge, TN, USA. He is in charge of the high performance middleware on top of Myrinet: MPI, VIA, SHMEM. His points of interest are high speed interconnects, message passing and high performance storage I/O.

IW - Myrinet ; SCSI ; Storage ; Disk I/O ; Linux ; Cluster
DT - FLA Full Length Article
TFT - EP Electricity-Physics
LA - EN
NLIN- 223